Head Linear Reduction

Vincent Danos
Équipe PPS - Université Paris 7 & CNRS
Vincent.Danos@pps.jussieu.fr

Laurent Regnier
Institut de Mathématiques de Luminy - Université Aix-Marseille 2 & CNRS
Laurent.Regnier@iml.univ-mrs.fr

June 7, 2004

Abstract

This paper defines *head linear reduction*, a reduction strategy of λ-terms that performs the minimal number of substitutions for reaching a head normal form. The definition relies on an extended notion of redex, and head linear reduction is therefore not a strategy in the exact usual sense. Krivine's Abstract Machine is proved to be sound by relating it both to head linear reduction and to usual head reduction. The first proof suggests a variant machine, the Pointer Abstract Machine, which is also proved to be sound with respect to head linear reduction.

1 Introduction

A common, and somewhat hidden notion, that one finds in various approaches to higher-order computation, such as geometry of interaction, game semantics, optimal reduction, and ordinary operational semantics, is that of *head linear reduction*. The purpose of this paper is bring to the center stage this notion, and explain how nicely it relates, and in some sense is implicit to the Krivine Abstract Machine (KAM). The KAM is a simple computational model for λ-calculus, that has been known to the authors back to Krivine's lectures on λ-calculus in 1987. The relation it has with head linear reduction has played a central role in their understanding of higher-order computation [1, 2, 3, 4, 5, 6], but never found its way to printed form.

Part of the reason why the notion of head linear reduction is not widespread, comes from the fact that it is not a reduction strategy in the usual sense. In order to define it properly, one has to extend the notion of redex and at the same time restrict to linear substitution (substitution of a single variable occurrence). Indeed, the concept of head linear reduction is the importation in λ-calculus of a computation strategy in linear logic proof-nets. In the more flexible and finer-grained syntax of proof nets, head linear reduction stands out as the most natural strategy (a typical usage can be found in the Mascari-Pedicini Böhm-like separation theorem [7] for pure proof nets). Translated back in the language of λ-calculus, one obtains a reduction that performs the minimal amount of substitutions necessary to find which occurrence of variable will stand in head position in the head normal form (if any). This is enough to define a notion of evaluation, or execution, as in a functional programming language, but not enough to always produce a head normal form.

Another reason why head linear reduction has not shown up earlier in the theory of λ-calculus, is that its associated notion of observational equivalence is the same as the one associated to usual head reduction. Thus, most models, with the notable exception of the intensional models of game semantics [8, 9], won't be able to tell the difference. The more recent differential λ-calculus [10] also has a strong relation to head linear reduction, in that this strategy effectively computes the differential terms that linearly approximate a given term.

Outline. Head linear reduction is constructed in the first part of the paper; the second part is devoted to the presentation and soundness proofs of the KAM; the third and last part introduces a variant of the

KAM, the Pointer Abstract Machine (PAM). This variant is even lazier than the KAM, in that closures are computed by need, just as the KAM performs substitutions only when necessary. As a consequence, it is faster than the KAM in many cases [11]. Soundness is proved by relating the PAM to head linear reduction in exactly the same way as with the KAM. This is key to the understanding that Hyland-Ong game semantics [9] is fundamentally a semantics of PAM computation traces [5], a fact which is basic to the recent work on *algorithmic* game semantics [12, 13, 14]. A number of interesting variants and extensions of the PAM have been studied by Herbelin and Curien [15].

2 Notations, conventions

Uppercase letters are used to denote terms and subterms. Lowercase letters such as x, y will stand for variables, while z, and subscripted xs and ys will denote *occurrences* of variables. We will consider λ-terms as trees consisting of:

- variable leaves labelled with a variable name,
- binary application nodes,
- and unary λ-nodes labelled with a variable name.

We denote terms with the standard notations: TU for the application of a term T to a term U, $\lambda x\, U$ for the abstraction of the variable x in the term U and $T[U/x]$ for T in which occurrences of the free variable x are substituted by U; additionally we allow free usage of parentheses to disambig(u)ate expressions.

An *application term* is a tree rooted at an application node, an *abstraction term* is a tree rooted at a λ-node. A *subterm* of a term T is a subtree of T. An *occurrence of variable* in T is a variable node in T, an *argument subterm* of T is a subterm that is the argument subtree of an application node in T.

We assume that all λ-terms respect Barendregt's convention, namely that each variable has a distinct name. For example we write $\delta\delta = (\lambda x\, xx)\lambda y\, yy$. In particular a λ-node is completely determined by the name of its variable, say x, and we will denote it just as λx. Also this will allow us to identify occurrences of variables in a term by their variable name together with an occurrence number, *e.g.*, in $\delta\delta$ the two occurrences of the variable x (resp. y) will be named x_1 and x_2 (resp. y_1 and y_2).

We will write $(a_1, \ldots, a_n)$ for the sequence $a_1, \ldots, a_n$, $()$ for the empty sequence, $a \cdot l$ for the sequence beginning with a and ending with l, $l \cdot l'$ for the concatenation of l and l', and $l \cdot a$ for the list beginning with l and ending with a.

3 Head linear reduction

For the sake of comparing with head linear reduction, to be defined shortly, we recall first the definition of head reduction. Any term T can be written uniquely as:

$$T = \lambda x_1 \ldots \lambda x_n\, U\, V_1 \ldots V_m$$

where U is either a variable, in which case T is said to be in *head normal form*, or a redex which is then called the *head redex* of T. The *head reduction* is the strategy that fires the head redex at each step, thus stopping when reaching a head normal form. When additionally, one asks that there is no prefixing binder, *i.e.*, that $n = 0$, one obtains the notion of *weak head reduction*, thus stopping either when reaching a term of the form $\lambda x\, U$ or $x\, V_1 \ldots V_m$.

A standard theorem in λ-calculus theory, that we will use later, is that as soon as T has a head normal form (is beta-equivalent to a head normal form) then its head reduction terminates.

To properly define head linear reduction, we first slightly generalize the notion of redex.

Head occurrence (hoc) and prime redex. The set of *spine subterms* of a term T is defined by induction on T:

- T is a spine subterm of T,
- and if $T = UV$ (resp. $T = \lambda x\, U$) then all the spine subterms of U are spine subterms of T.

Any term T has exactly one spine subterm which is an occurrence of variable, denoted $v_h(T)$ and called the *head occurrence* (hoc) of T.

The *head λ list* $\lambda_h(T)$ of T is a sequence $(\lambda x_1, \ldots, \lambda x_n)$ of abstraction subterms of T; the *prime redexes* are pairs $(\lambda x, A)$ where λx is an abstraction subterm of T and A is an argument subterm of T, called the *argument of the prime redex*. Both are defined by induction on T:

- if T is a variable then $\lambda_h(T)$ is the empty sequence and T has no prime redexes;
- if $T = UV$ then either $\lambda_h(U)$ is empty in which case $\lambda_h(T)$ is defined to be empty and the prime redexes of T are those of U; or $\lambda_h(U)$ begins with λx, that is $\lambda_h(U) = (\lambda x) \cdot l$, and we set $\lambda_h(T) = l$ and define the prime redexes of T to be those of U with the addition of $(\lambda x, V)$;
- if $T = \lambda x\, U$ then $\lambda_h(T) = (\lambda x) \cdot \lambda_h(U)$ and the prime redexes of T are those of U.

Suppose the hoc of T is an occurrence of the (bound) variable x; then the prime redex $(\lambda x, V)$ if it exists is called the *hoc redex* of T.

This definition of hoc redex is the very reason for the generalization of the notion of redex: indeed in standard λ-calculus the binder of the hoc is not in general in redex position, as for example in $(\lambda x\, \lambda y\, y\, U)VW$ where the hoc redex $(\lambda y, W)$ is not a standard one; in this case, head reduction will begin by firing the regular redex $(\lambda x, V)$ yielding the term $(\lambda y\, yU')W$. In this latter term the hoc redex and the head redex now coincide. The lemma below slightly generalizes this example.

Let T be a term and $r = (\lambda x, V)$, $s = (\lambda y, W)$ be two prime redexes in T. We say that r *contains* s if the node λy lies in the scope of the node λx, that is if λy is located in the subtree rooted at λx. We say that r and s are *consecutive* if r contains s and no prime redex t other than r and s is contained in r and contains s.

Lemma 1 *Let r_0, …, r_p be a sequence of consecutive prime redexes in T such that r_0 is contained in no prime redexes in T. Then r_0 is the head redex of T and in exactly p head reduction steps, r_p will become the head redex of the (reduct of) T.*

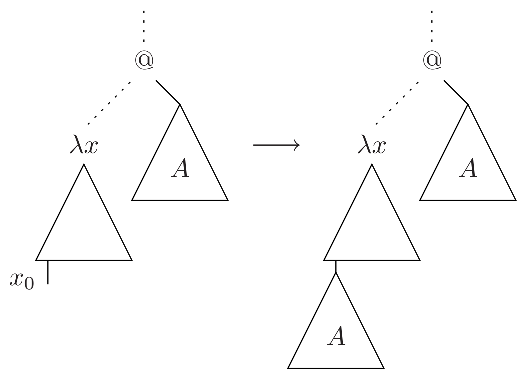
This is an immediate consequence of the definition. Theorem 2 below uses this lemma.

If one considers λ-terms up to redex permutations, as in the so-called σ-equivalence [16], then prime redexes are exactly spine redexes.

A term T contains at most one hoc redex. In the case where T contains no hoc redex we say that T is in *quasi-head-normal form* (qhn).

Head linear reduction. Let T be a term with hoc x_0, not in qhn form, and let $(\lambda x, A)$ be its hoc redex, the head linear reduct of T is obtained by substituting the occurrence x_0 with A. Substituting A involves α-conversion of all variables bound in A to preserve Barendregt's convention.

Let us insist that in this transformation, *only* the occurrence x_0 of x is substituted, and that $(\lambda x, A)$ is *not* consumed in the process (see the figure below). In particular, the reduct of T is always bigger than T (but not always strictly so, as A can be an occurrence of variable). This transformation clearly defines a map from the reduct subterms to the subterms of T called below the *residual* map.



As an example here are the first few steps of the head linear reduction of $\delta\delta$. At each step the hoc redex is highlighted by underlining the corresponding λ and argument, and the hoc is boxed. We perform on the fly α-conversion when copying $\lambda y\, yy$.

$$\begin{aligned}
& (\underline{\lambda} x \boxed{x} x)\underline{\lambda y\, yy} \\
\longrightarrow\quad & (\lambda x\,(\underline{\lambda} y' \boxed{y'} y')\underline{x})\lambda y\, yy \\
\longrightarrow\quad & (\underline{\lambda} x\,(\lambda y' \boxed{x} y')x)\underline{\lambda y\, yy} \\
\longrightarrow\quad & (\lambda x\,(\lambda y'\,(\underline{\lambda} y'' \boxed{y''} y'')\underline{y'})x)\lambda y\, yy \\
\longrightarrow\quad & (\lambda x\,(\underline{\lambda} y'\,(\lambda y'' \boxed{y'} y'')y')\underline{x})\lambda y\, yy \\
\longrightarrow\quad & (\underline{\lambda} x\,(\lambda y'\,(\lambda y'' \boxed{x} y'')y')x)\underline{\lambda y\, yy} \\
\longrightarrow\quad & (\lambda x\,(\lambda y'\,(\lambda y''\,(\underline{\lambda} y''' \boxed{y'''} y''')\underline{y''})y')x)\lambda y\, yy
\end{aligned}$$

Theorem 2 *If T' is obtained from T by head linear reduction, then T and T' are β-equivalent.*

If T is in qhn form and has n prime redexes, then the head reduction of T leads to a head normal form in exactly n steps.

If T is any term, the head linear reduction of T terminates iff the head reduction of T terminates.

Proof. The first statement is proved by induction on the length (the number of steps) of the head linear reduction leading from T to T'; when T' is obtained by one step of head linear reduction, we prove by induction on the number of consecutive prime redexes from the root of T to its hoc redex, that T and T' both reduce to the same term by head reduction, which is easily seen thanks to lemma 1 and to the remark that the first consecutive prime redexes up to the hoc redex of T are the same in T and T'.

The second statement comes by induction on n and considering that in a qhn form, no β-reduction can substitute the hoc, thus no β-reduction can create a new prime redex.

As for the last statement, if we suppose that the head linear reduction terminates, then it yields a qhn form, thus according to the second statement, a head normal form; therefore the term has a head normal form from which one deduces by standard λ-calculus theory that its head reduction terminates.

The converse follows from the KAM theorems 4 and 8 below. $\square$

Theorem 3 (Residual invariance) *Let T be a term, and T' be obtained from T by head linear reduction, all arguments in the successive hoc redexes leading to T' are (residuals of) subterms of T.*

Proof. Straightforward. $\square$

This property is characteristic of head linear reduction, that is to say that it is not true of β-reduction, not even head reduction. For example consider $T = (\lambda x\, x(xU))\lambda y\, y$ with x not in U; if one fires the head redex, one gets $(\lambda y\, y)(\lambda y' y')U$, where the new hoc redex argument $(\lambda y' y')U$, is not a subterm of T.

Substitution sequence. Let $T = T_0$ be a term and $(T_0, T_1, \dots)$ the (possibly infinite) sequence of its successive reducts by head linear reduction. We call *substitution sequence* of T the sequence $((z_0, A_1), (z_1, A_2), \dots)$ where the z_i's and the A_i's are respectively occurrences of variable in T and argument subterms of T such that for each i the hoc of T_i is a residual of z_i and the argument of the hoc redex in T_i is a residual of A_{i+1}.

In the $\delta\delta$ example above, the substitution sequence reads

$$((x_1, \lambda y\, yy), (y_1, x_2), (x_2, \lambda y\, yy), (y_1, y_2), (y_2, x_2), (x_2, \lambda y\, yy), (y_1, y_2), (y_2, y_2), (y_2, x_2), (x_2, \lambda y\, yy), \dots)$$

Note in particular that the subterms xx and yy will never appear in this sequence.

4 Krivine's Abstract Machine (KAM)

There are many ways to present the KAM; in his original presentation [17], Krivine used super combinators and double de Bruijn indices to access values in the environments. Also one may avoid use of de Bruijn indices altogether and encode environments as associative tables accessed by variable names; Sylvain Lippi has shown in his thesis [18] that this approach was correct, and surprisingly enough, that α-conversion wasn't necessary.

De Bruijn notation. Here we shall give the simplest presentation, using simple de Bruijn indices. So in this section all λ-terms are assumed to be indexed. More precisely, each variable node is labelled not only with a variable name, but also with a positive integer: the *index* of the occurrence. If the occurrence is bound in the term, its index counts exactly the number of λ-nodes lying between the variable node and its binder node (in $\lambda x\, x$ the occurrence of x has index 0). If the occurrence is free then its index i is greater (or equal) than the number l of λ-nodes lying between the variable node and the root of the term. We will call *global index* of the node the difference $i - l$.

Closures and states. The KAM is an abstract machine defined by a set of *states* and a set of *transitions* between states. The *execution* is a sequence of transitions, starting from some initial state and ending on some final state.

For each integer k, we define the *closures* of depth $\leq k$ to be the pairs (T, ρ) where T is a term and either $\rho = ()$ is empty, or $k > 0$, and $\rho = (c_1, \ldots, c_n)$ is a finite sequence of closures of depth $\leq k - 1$.

A finite sequence of closures will be called an *environment* or a *stack*, depending on its role in the KAM.

A KAM state s is a triple (T_s, ρ_s, σ_s) where T_s is a λ-term, ρ_s is an environment (so that (T_s, ρ_s) actually is a closure) and σ_s is another environment called the stack of s (also called the *continuation*, in particular when implementing λ-μ-calculus). Depending on the shape of T_s, we will say that s is a variable, a λ or an application state.

With this presentation, a KAM state is a tree. The drawback is that implementations following this scheme are kind of wasteful with memory and rely on the underlying garbage collector of the implementing language. One could follow another scheme where a state is actually a dag implementing sharing of environments, thus making sparser use of memory. This is the natural way to implement the KAM in a language with pointers. Here we shall stick to our simple presentation with states as trees.

From states to terms. States encode the successive reducts of a term during the execution of the KAM by a mechanism related to explicit substitution. If $s = (T_s, \rho_s, \sigma_s)$ is a state then ρ_s will contain the value of the free variables of T_s, and σ_s is the stack of arguments (in closures) of T_s. This may be formalized by defining the *expansion* $\exp(s)$ of s to be the term:

$$\exp(s) = [\![T_s]\!]_{\rho_s} [\![T_1]\!]_{\rho_1} \ldots [\![T_n]\!]_{\rho_n}$$

where $\sigma_s = ((T_1, \rho_1), \ldots, (T_n, \rho_n))$ and $[\![T]\!]_\rho$ is the *expansion of T in the environment* ρ defined as follows:

$$[\![z]\!]_{((T_0,\rho_0),\ldots,(T_n,\rho_n))} = \begin{cases} [\![T_i]\!]_{\rho_i} & \text{if } z \text{ has index } i \leq n \\ z & \text{otherwise} \end{cases}$$

$$[\![UV]\!]_\rho = [\![U]\!]_\rho [\![V]\!]_\rho$$

$$[\![\lambda x\, U]\!]_\rho = \lambda \bar{x}\, [\![U]\!]_{(\bar{x},()) \cdot \rho}$$

where $\bar{x}$ is a fresh variable name.

Transitions. A transition $(s \longrightarrow s')$ is a transformation on states mapping a *source* state s onto a *target* state s'. Given a state $s = (T_s, \rho_s, \sigma_s)$, at most one transition may apply. One says the transition system is *deterministic*. Which transition applies depends only on the shape of T_s:

Term	Environment	Stack		Term	Environment	Stack
UV	ρ	σ	$\longrightarrow$	U	ρ	$(V, \rho) \cdot \sigma$
$\lambda x\, U$	ρ	$c \cdot \sigma$	$\longrightarrow$	U	$c \cdot \rho$	σ
z	$((T_0, \rho_0), \ldots, (T_n, \rho_n))$	σ	$\longrightarrow$	T_z	ρ_z	σ

where z is the de Bruijn index of an occurrence of variable.

Termination. There are two kinds of states where no transition is defined:

- $(\lambda x\, U, \rho, ())$: the term is an abstraction but the stack is empty.
- $(z, (c_0, \dots, c_n), \sigma)$ where $n < z$; the term is a variable occurrence with de Bruijn index z but the environment contains less than $z + 1$ elements.

As will be clear from the statement of correctness, these two kinds of termination happen when the term being executed normalizes to some term beginning with a λ (first case) or a free variable applied to some arguments (second case).

Run. The *run* of a term T (the T-run), is the possibly infinite, maximal sequence of states $(s_0, \dots)$ such that $s_0 = (T, (), ())$ (empty environment and stack), and for each i, s_{i+1} results from s_i by applying the appropriate transition.

A first way to express the correctness of the KAM is to relate it to weak head reduction.

Theorem 4 (Correctness w.r.t. weak head reduction) *Let s be a state appearing in a T-run; then T reduces to $\exp(s)$ by weak head reduction. Conversely, if T' is a reduct of T by weak head reduction, then there is a state s in the T-run such that $\exp s = T'$.*

Proof. (Sketch, a detailed proof is in Krivine's paper in the same volume [19].) The only transition that changes $\exp(s)$ is the λ-transition. Before a λ-transition the state is $s = (\lambda x\, U, \rho, (V, \rho') \cdot \sigma)$, and one has:

$$\exp(s) = (\lambda \bar{x}\, [\![U]\!]_{(\bar{x},()) \cdot \rho})[\![V]\!]_{\rho'} V_1 \dots V_n$$

where the V_i's are the expansions of the elements of σ. The transition yields $s' = (U, (V, \rho') \cdot \rho, \sigma)$ the expansion of which is:

$$\exp(s') = [\![U]\!]_{(V,\rho') \cdot \rho} V_1 \dots V_n.$$

To see that $\exp(s')$ actually is the reduct by firing the head redex of $\exp(s)$, one needs the following substitution lemma:

$$[\![U]\!]_{(V,\rho') \cdot \rho} = [\![U]\!]_{(\bar{x},()) \cdot \rho}\big[[\![V]\!]_{\rho'}/\bar{x}\big]$$

□

In view of this theorem, it seems that the KAM implements weak head reduction. Actually the situation is a bit subtler as one sees from the argument above. The machine λ-transition is very simple, and in particular doesn't involve any copying, whereas the corresponding transformation on expansions fires the head redex. On the other hand, the machine application transition is not so simple, and involves duplication of the environment (or in a shared environment implementation, at least the creation of a new environment cell); also variable transitions are rather complex, as they have to fetch the value of the variable in the environment. However for both these transitions, the corresponding transformations on expansions are identities.

Expansion is performing all the substitutions corresponding to the head redexes that have been recorded, but precisely the KAM just records head redexes in the environment, and delays the actual substitutions until they are really needed, that is until one reaches a variable transition.

As we will see now, the KAM relates in a simpler way to head linear reduction, and accordingly correctness with respect to head linear reduction is much simpler to state.

Substitution sequence of a run. The *substitution sequence of a T-run* is the possibly infinite sequence $((z_0, A_1), \dots)$ where z_i (resp. A_{i+1}) is the occurrence of variable (resp. the term) appearing in the source state (resp. the target state) of the ith variable transition.

This definition parallels the definition of substitution sequence of a head linear reduction, and just as head linear reduction does, the KAM has the residual invariance property that all terms appearing in the states of a T-run are subterms of T. Moreover, because closures are created only by the application transitions, all the terms appearing in closures are argument subterms of T. Which means that all the A_i's for $i > 0$ are argument subterms of T.

Theorem 5 (Derivation of a run) *Let T be a term, T' be its reduct by one step of head linear reduction, $(\lambda x, A)$ be the hoc redex of T, and x_0 be the hoc of T. Then either the KAM substitution sequence of T is empty, or it has the form $(x_0, A) \cdot l$ where l is the KAM substitution sequence of T'.*

Proof. By hypothesis the term T has the shape $\ldots(\ldots \lambda x \ldots x_0 \ldots)A\ldots$. The successive transitions of the T-run until the first variable transition will be as follows:

$$\begin{array}{cccc}
1 & T, & (), & () \\
\vdots & \vdots & \vdots & \vdots \\
2 & (\ldots \lambda x \ldots x_0 \ldots)A, & \rho_1, & \sigma_1 \\
3 & \ldots \lambda x \ldots x_0 \ldots & \rho_1, & (A, \rho_1) \cdot \sigma_1 \\
\vdots & \vdots & \vdots & \vdots \\
4 & \lambda x \ldots x_0 \ldots, & \rho_2 \cdot \rho_1, & (A, \rho_1) \cdot \sigma_1 \\
5 & \ldots x_0 \ldots, & (A, \rho_1) \cdot \rho_2 \cdot \rho_1, & \sigma_1 \\
\vdots & \vdots & \vdots & \vdots \\
6 & x_0, & \rho_3 \cdot (A, \rho_1) \cdot \rho_2 \cdot \rho_1, & \sigma_2 \\
7 & A, & \rho_1 & \sigma_2
\end{array}$$

where ρ_1 is an environment containing values for the variables free in A. Until step 7, only application and abstraction transitions are applied, thus the environment may only grow. Also, thanks to our hypothesis that $(\lambda x, A)$ is a prime redex, we know that the application of step 2 matches the λ in step 4 which is why we get the same stack at step 3 and 4. Finally the length of the environment ρ_3 is the number of λs encountered between steps 5 and 6, that is between λx and x_0; as x_0 is an occurrence of the variable x, its de Bruijn index is precisely this same number, which is why we get A in step 7.

Now let T' be the reduct of T by one step of head linear reduction, thus T' reads $\ldots(\ldots \lambda x \ldots \bar{A} \ldots)A\ldots$ where $\bar{A}$ is the copy of A that x_0 is substituted with; in particular all de Bruijn indices corresponding to occurrences of free variables in A are *lifted* by the number of λs between the application to A and x_0; this number is precisely the length of the environment $\rho_3 \cdot (A, \rho_1) \cdot \rho_2$, so that the values of the free variables of A in ρ_1 are exactly the same as the values of the free variables of $\bar{A}$ in $\rho_3 \cdot (A, \rho_1) \cdot \rho_2$. We'll comment later on this crucial point, but for now let us look at the first transitions in the T'-run:

$$\begin{array}{cccc}
1 & T', & (), & () \\
\vdots & \vdots & \vdots & \vdots \\
2 & (\ldots \lambda x \ldots \bar{A} \ldots)A, & \rho_1, & \sigma_1 \\
3 & \ldots \lambda x \ldots \bar{A} \ldots & \rho_1, & (A, \rho_1) \cdot \sigma_1 \\
\vdots & \vdots & \vdots & \vdots \\
4 & \lambda x \ldots \bar{A} \ldots, & \rho_2 \cdot \rho_1, & (A, \rho_1) \cdot \sigma_1 \\
5 & \ldots \bar{A} \ldots, & (A, \rho_1) \cdot \rho_2 \cdot \rho_1, & \sigma_1 \\
\vdots & \vdots & \vdots & \vdots \\
6 & \bar{A}, & \rho_3 \cdot (A, \rho_1) \cdot \rho_2 \cdot \rho_1, & \sigma_2
\end{array}$$

We are to show that the substitution sequences generated from (A, ρ_1, σ_2) and $(\bar{A}, \rho_3 \cdot (A, \rho_1) \cdot \rho_2 \cdot \rho_1, \sigma_2)$ are identical. To do so, we introduce a notion of *abstract closure*: an abstract closure of depth 0 is a (non indexed) term, and for $k > 0$, an abstract closure of depth $\leq k$ is a pair (T, α) where T is a (non indexed) term and α, the *value function*, is a partial function from the set of occurrences of free variables in T into abstract closures of depth $< k$.

To each closure $c = (T, \rho)$ of depth $\leq k$, we associate an abstract closure $\underline{c}$ of depth $\leq k$ called the *abstraction of* c: if $k = 0$, which means that ρ is the empty environment, then $\underline{c}$ is just T with de Bruijn indexing removed. Otherwise $\rho = (c_0, \ldots, c_n)$ for some closures $c_0, \ldots c_n$ of depth $< k$. Let $\underline{c}_0, \ldots \underline{c}_n$ be their respective abstractions. Then $\underline{c}$ is the term T with de Bruijn indexing removed, together with the value function associating to each occurrence of free variable in T with *global* index $i \leq n$, the abstract closure $\underline{c}_i$.

We will say that two closures c and c' are *equivalent* if they have the same abstraction.

Lemma 6 *Let $c = (T, \rho)$ and $c' = (T', \rho')$ be two equivalent closures and let α be their common value function. Then up to de Bruijn indexing T and T' are equal. Thus α is defined on the occurrences of T as well as on the occurrences of T'. For each pair (z, z') of corresponding occurrences of variable in respectively T and T', either both are not in the domain of α, or both have the same value.*

This is immediate from the definition of a closure abstraction.

Let now $s = (T, \rho, \sigma)$ and $s' = (T', \rho', \sigma')$ be two states of the KAM. We will say that s and s' are equivalent if the closures (T, ρ) and (T', ρ') are equivalent, $\sigma = (c_1, \ldots, c_n)$ and $\sigma' = (c'_1, \ldots, c'_n)$ have the same length n and for each $i = 1, \ldots n$, c_i and c'_i are equivalent. Here comes the key lemma:

Lemma 7 *Let s_0 and s'_0 be two equivalent KAM states, $r = (s_0, s_1 \ldots)$ and $r' = (s'_0, s'_1 \ldots)$ be the two KAM runs starting respectively from s_0 and s'_0. Then for each n the states s_n and s'_n are equivalent.*

We show that s_n and s'_n are equivalent by induction on n. It is our hypothesis when $n = 0$. In the induction case we first remark that $s_n = (T_n, \rho_n, \sigma_n)$ and $s'_n = (T'_n, \rho'_n, \sigma'_n)$ being equivalent, T_n and T'_n are the same term up to indexing; thus the same type of transition (application, abstraction or variable) will apply to both states. The interesting case is when both T_n and T'_n are occurrences of variable z and z'.

Let α be the common value function associated to the closures (z, ρ_n) and (z', ρ'_n) and i be the index of z. Note that i is also the global index of z in the term z. If i is greater or equal than the length of ρ_n then α has empty domain. Thus z' is not in the domain of α which means that its index i' is also greater or equal than the length of ρ'_n. In this case both KAM runs stop at the step n, because no variable transition applies.

If i is strictly smaller than the length of ρ_n, then by definition of the abstraction, z is in the domain of α and the value of z is the abstraction of c_i, the ith element of ρ_n. By the lemma above, z' must be in the domain of α, which means that its index i' is strictly smaller than the length of ρ'_n and its value is the abstract closure associated to $c'_{i'}$, the i'th closure in ρ'_n.

Now by definition of the KAM, the states s_{n+1} and s'_{n+1} are respectively $(T_{n+1}, \rho_{n+1}, \sigma_n)$ and $(T'_{n+1}, \rho'_{n+1}, \sigma'_n)$ where $c_i = (T_{n+1}, \rho_{n+1})$ and $c'_{i'} = (T'_{n+1}, \rho'_{n+1})$. As c_i and $c'_{i'}$ have the same abstraction, we see that s_{n+1} and s'_{n+1} are equivalent. Thus the lemma is proved.

Back to our two states (A, ρ_1, σ_2) and $(\bar{A}, \rho_3 \cdot (A, \rho_1) \cdot \rho_2 \cdot \rho_1, \sigma_2)$, from our computation above one easily deduces that they are equivalent. Thus the subsequent KAM runs are equivalent and the theorem follows as we already know. □

As a consequence, there is a one-to-one correspondence between substitution steps in the weak head linear reduction of T and the variable transitions in the T-run.

Corollary 8 *For all terms T, the KAM substitution sequence of T and the weak head linear reduction substitution sequence of T coincide.*

5 The Pointer Abstract Machine (PAM)

So head linear reduction and the KAM produce the same substitution sequences. Implementationwise, both can be criticized. Head linear reduction uses α-conversion, and produces ever bigger terms, making it unreasonable for a real implementation, or even for paper calculations. The KAM is already better, but even if it performs only necessary substitutions (in variable transitions), it also performs many operations which may prove useless for the rest of the computation. We have just seen an instance of this in the proof above, where the T' run begins with the construction of a big environment that is mostly useless for the rest of the computation as only the ρ_1 part of it will be ever used.

We will now define a new machine, the Pointer Abstract Machine (PAM), focusing on substitution sequences; the basic idea is to drop the λ and application transitions and to compute the value of an occurrence of variable only when reaching it. For this to be possible, each occurrence of variable in a PAM run has to come together with a *pointer*, indicating where in the run the closure giving its value could have been built (but was not), that is, where the (occurrence of) binder of this variable has been encountered. No closure will ever be built. A run will be exactly that, a sequence of occurrences of variables with pointers.

This simple idea results in a mechanism which is streamlined enough, so that one can inspect its mathematical structure, and turn it into a well-structured model of computation, the Hyland-Ong game model [9]. However, the machine was formulated in 1988, and never really elaborated as a semantics. Comparable ideas were developed by Coquand at the same time [20]. The game model was developed independently, and the model-machine relationship only understood later [5]. Most of the progresses in game semantics since, have been related to the classification of how much a given abstract term, or a strategy, is allowed to know about its own PAM run when interacting against an unknown environment.

5.1 Preliminaries

Before defining the PAM we need to introduce some new specific terminology, some of it (justification, move, etc.) borrowed from game semantics.

Justification. Let T be a term. An argument subterm of a term T will be called a *spine argument* if it is the argument of some spine application node of T.

If z is an occurrence of variable in T, let T_z be the maximal subterm of T having z as hoc. Note that T_z is either T itself (if z is the hoc of T), or an argument subterm of T. Let A be an argument subterm of T; we say that A is *justified* by z if A is a spine argument in T_z. This argument subterm A may or may not be an argument of a prime redex of T_z. When it is not, we say that A *is an argument of* z.

Consider for instance the term $T = (\lambda x\, \lambda y\, x_0 U_1)U_2 U_3 U_4$, then $T = T_{x_0}$, all U_is are justified by the hoc x_0, U_1, U_4 are arguments of x_0, and U_2, U_3 are respectively argument of λx and λy, that is $(\lambda x, U_2)$ and $(\lambda y, U_3)$ are prime redexes.

Remark that the set of arguments of an occurrence of variable z is a sequence (reading the term from left to right) so we may use expressions like "the ith argument of z"; in the example above U_1 and U_4 are respectively the first and the second argument of x_0.

We call *arity* of an occurrence of variable in T the number of its arguments. In the example above, the arity of x_0 is 2. It is important to note that, because terms are untyped, arity is associated to *occurrences* of variable, *e.g.*, in $\lambda x\, x_1 x_2$, x_1 has arity 1 while x_2 has arity 0.

Binding. We say that a variable x in T is *bound in the subterm* A, or that A binds x, if A is the maximal subterm of T having λx among its spine λs. Note that A is either T itself, or an argument subterm of T. If $(\lambda x_1, \dots \lambda x_k)$ is the head λ list of A, then we say that λx_i *has rank* i in A. Note also that λ-nodes that are part of a prime redex of A have no rank.

Subterm chain. A *chain* in a term T is a sequence $s = (A_0, \dots, A_k)$ of subterms of T such that $A_0 = T$ and for each i, A_{i+1} is a spine argument in A_i. If z is the hoc of A_k, we say that s *is the chain of* z.

Lemma 9 *Each occurrence of variable z has a unique subterm chain. As a consequence, z is bounded in T iff there is one A_i in s that is binding z.*

Proof. Basically the subterm chain of z is the path from the root of the term to z, which is unique because T is a tree. □

PAM state. A *pointed sequence* of the term T is a finite sequence of *moves*: $((p_1, z_1, A_1), \dots, (p_n, z_n, A_n))$ where for each i we have: $1 \leq p_i \leq i$, z_i is an occurrence of variable in T, A_i is an argument subterm of T. A *PAM state* for T is a pointed sequence $((p_1, z_1, A_1), \dots, (p_n, z_n, A_n))$ satisfying:

- z_1 is the hoc of T and for each $i > 1$, z_i is the hoc of A_{i-1};
- A_i is justified by z_{p_i} (we call this the *application constraint*).

Back to our prefered example, figure 1 shows a PAM state for $\delta\delta = (\lambda x\, xx)\lambda y\, yy$ (actually the beginning of the PAM run).

Lemma 10 *Let $s = ((p_1, z_1, A_1), \dots, (p_n, z_n, A_n))$ be a PAM state for a term T, z be the hoc of A_n. For each $1 \leq i \leq n$, define $\phi_s(i) = p_i - 1$ and let k be the smallest integer such that $\phi_s^k(n) = 0$. Then, reading $A_0 = T$, the sequence $(A_{\phi_s^k(n)}, \dots, A_{\phi_s(n)}, A_n)$ is the subterm chain of z in T.*

$$\begin{array}{ll}
1. & (1, x_1, \lambda y\, yy) \\
2. & (1, y_1, x_2) \\
3. & (1, x_2, \lambda y\, yy) \\
4. & (2, y_1, y_2) \\
5. & (1, y_2, x_2) \\
6. & (1, x_2, \lambda y\, yy) \\
7. & (4, y_1, y_2) \\
8. & (2, y_2, y_2) \\
9. & (1, y_2, x_2) \\
10. & (1, x_2, \lambda y\, yy)
\end{array}$$

Figure 1: Beginning of the PAM run of $(\lambda x\, xx)\lambda y\, yy$

Proof. First, since $p_i \leq i$, we have $\phi_s(i) < i$ for $1 \leq i \leq n$ so that k is well defined. Now the lemma follows by considering that for each i, A_i is a spine argument in $A_{\phi_s(i)}$; this is because by definition of a PAM state, A_i is justified by z_{p_i} and z_{p_i} is the hoc of $A_{p_i-1} = A_{\phi_s(i)}$. □

The sequence of indices $(0, \phi_s^{k-1}(n), \dots, \phi_s(n), n)$ corresponds to the notion of *view* in game semantics.

5.2 PAM Transitions

Let T be a term and $s = ((p_1, z_1, A_1), \dots, (p_n, z_n, A_n))$ be a PAM state for T. If $n = 0$, that is s is empty, and T is not a qhn, then the first transition initializes s with the move $(1, z_1, A_1)$ where z_1 is the hoc of T and A_1 is the argument of the hoc redex of T. Otherwise we build a new move $(p_{n+1}, z_{n+1}, A_{n+1})$ in the following way:

1. set z_{n+1} to be the hoc of A_n;
2. we are to find the binder of z_{n+1}. If z_{n+1} is free, then no transition applies. Otherwise we know that its binder lies in the chain of z_{n+1}, thus the lemma above tells us that there is a k such that $A_{\phi_s^k(n)}$ binds z_{n+1} (as before denoting $A_0 = T$). Let $j = \phi_s^k(n)$, so that $0 \leq j \leq n$, and let x be the variable z is an occurrence of. There are two cases:
   (a) λx is not a head λ of A_j, that is, A_j has a prime redex $(\lambda x, A)$. Then we set $A_{n+1} = A$ and $p_{n+1} = j + 1$; indeed z_{j+1} is the hoc of A_j, thus justifies $A = A_{n+1}$.
   (b) λx is a head λ of A_j of rank r. We are now to find what goes into that λ. The idea is that A_j has been found as the value of z_j, that is in head linear reduction A_j would be substituted in z_j. What we do now depends on whether z_j has arity greater or equal to r:
      Yes: we define A_{n+1} to be the rth argument of z_j, $p_{n+1} = j$ and we are done (in particular we satisfy the application constraint of PAM states);
      No: we have to look further up in s for the value; if $j = 0$ no transition applies. Otherwise we replace r by $r - a + l$ where a is the arity of z_j, and l is the number of head λs in A_{j-1}; we also replace j by $j - 1$ and repeat the process.

Case (2a) typically arises when $j = 0$, which means that we are looking for the value of an occurrence of variable that is bound in a prime redex of T. However when T contains inner redexes it may be that $j = n$, in which case $p_{n+1} = n + 1$.

The relatively complex computation done in case (2b) is the price to pay for having no stacks or environments. One has to do a bit of gymnastics to find the argument subterm that matches λx. On the other hand, no time is spent constructing closures that will never be used. The correctness proof, which we are not giving here, essentially consists in understanding that PAM-transitions do these on-the-fly value fetching steps correctly.

Theorem 11 (PAM correctness) *Let $r = ((p_1, z_1, A_1), \dots)$ be a PAM run of the term T, that is the (possibly infinite) sequence of moves obtained by repeatedly applying PAM transitions. Then the sequence $((z_1, A_1), \dots)$ is the substitution sequence of T.*

To end up this brief presentation of the PAM, let us mention the interesting following property:

Theorem 12 *Let T be a simply-typed closed lambda term with type $\perp$ (using constants of base type $\perp$) and suppose T is η-long. Let $((p_1, z_1, A_1), \ldots, (p_n, z_n, A_n))$ be the PAM-run of T. Then for each $1 \leq i \leq n$ we have:*

- *A_i is justified by z_{p_i};*
- *z_i is bound in A_{p_i}.*

The first item is just our application constraint; the second one is a dual property and follows from the fact that in an η-long term, the number of arguments of each occurrence of variable z is given by the type of z, thus is the same as the number of head lambdas of any subterm that may be substituted to z; in particular, when running η-long terms, in case (2b), one always has that z_j has arity greater or equal to r. This fact somewhat simplifies the structures of runs and brings them very close to being a semantics of computation traces [5] as said at the beginning of this section.

References

[1] Vincent Danos and Laurent Regnier. Local and asynchronous beta-reduction. In *Proceedings of the 8^{th} Symposium on Logic in Computer Science*. IEEE Computer Society Press, 1993.

[2] Andrea Asperti, Vincent Danos, Cosimo Laneve, and Laurent Regnier. Paths in the lambda-calculus. In *Proceedings of the 9^{th} Symposium on Logic in Computer Science*, Paris, 1994. IEEE Computer Society Press.

[3] Vincent Danos and Laurent Regnier. Proof-nets and the Hilbert space. In Jean-Yves Girard, Yves Lafont, and Laurent Regnier, editors, *Advances in Linear Logic*, volume 222 of *London Mathematical Society Lecture Note Series*. Cambridge University Press, 1995.

[4] Vincent Danos and Laurent Regnier. Reversible, irreversible and optimal lambda-machines. In Jean-Yves Girard, Mitsu Okada, and André Scedrov, editors, *Proceedings of 1996 Workshop on Linear Logic*, volume 3 of *Electronic Notes in Theoretical Computer Science*, Tokyo, 1996. Elsevier.

[5] Vincent Danos, Hugo Herbelin, and Laurent Regnier. Games semantics and abstract machines. In *Proceedings of the 11^{th} Symposium on Logic in Computer Science*, New Brunswick, 1996. IEEE Computer Society Press.

[6] Vincent Danos, Marco Pedicini, and Laurent Regnier. Directed virtual reduction. In Dirk van Dalen, editor, *Proceedings of CSL'96*, 1996.

[7] Gianfranco Mascari and Marco Pedicini. Head linear reduction and pure proof net extraction. In *Selected papers of the Meeting on the Mathematical Foundations of Programming Semantics (MFPS'92, Oxford)*, volume 135, pages 111–137. Theoretical Computer Science, 1994.

[8] Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. Full abstraction for PCF (extended abstract). In Masami Hagiya and John C. Mitchell, editors, *Theoretical Aspects of Computer Software. International Symposium TACS'94*, number 789 in Lecture Notes in Computer Science, pages 1–15, Sendai, Japan, April 1994. Springer-Verlag.

[9] Martin Hyland and Luke Ong. On full abstraction for PCF. *Information and Computation*, 163:285–408, 2000.

[10] Thomas Ehrhard and Laurent Regnier. The differential lambda-calculus. *Theoretical Computer Science*, 309(1-3):1–41, 2003.

[11] Hugo Herbelin. Private Communication, 1997.

[12] Pasquale Malacaria and Chris Hankin. Non-deterministic games and program analysis: An application to security. In *Logic In Computer Science*, Trento, 1999. IEEE Computer Society Press.

[13] Dan R. Ghica and Guy McCusker. Reasoning about idealised algol using regular languages. In *Proceedings, Twenty-Seventh International Colloquium on Automata, Languages and Programming*, Geneva, 2000. European Association for Theoretical Computer Science.

[14] S. Abramsky, D. R. Ghica, A. S. Murawski, and C.-H. L. Ong. Applying game semantics to compositional software modeling and verification. In *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, volume 2988 of *Lecture Notes in Computer Science*, pages 421–435. Springer-Verlag, 2004.

[15] Pierre-Louis Curien and Hugo Herbelin. Computing with abstract Böhm trees. In Masahiko Sato and Yoshihito Toyama, editors, *Proceedings of the Third Fuji International Symposium on Functional and Logic Programming (FLOPS'98)*, pages 20–39. World Scientific, 1998.

[16] Laurent Regnier. Une équivalence sur les lambda-termes. *Theoretical Computer Science*, 126:281–292, 1994.

[17] Jean-Louis Krivine. Un interpréteur du lambda-calcul. Manuscript, 1986.

[18] Sylvain Lippi. *Théorie et pratique des réseaux d'interaction (Interaction nets).* Thèse de doctorat, Université Aix-Marseille 2, 2002.

[19] Jean-Louis Krivine. A call-by-name lambda-calculus machine. *Higher Order and Symbolic Computation*, 2004.

[20] Thierry Coquand. A semantics of evidence for classical arithmetic. *Journal of Symbolic Logic*, 60(1):325–337, March 1995.